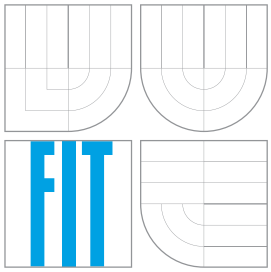
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DETEKCE PODOBNOSTI ZDROJOVÝCH SOUBORŮ V JAZYCE C

C LANGUAGE SOURCE FILES SIMILARITY DETECTION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
PETR REK

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. PETER MATULA

BRNO 2016

Zadání bakalářské práce/17996/2015/xrekpe00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Rek Petr**
Obor: Informační technologie
Téma: **Detekce podobnosti zdrojových souborů v jazyce C**
C Language Source Files Similarity Detection
Kategorie: Překladače

Pokyny:

1. Studujte problematiku zpětného inženýrství. Zaměřte se na zpětný překlad binárního kódu do vyšší formy reprezentace.
2. Seznamte se s překladačem Clang a jeho podporou pro vytváření nástrojů analyzujících kód v jazyce C.
3. Seznamte se se zpětným překladačem společnosti AVG a současnou implementací nástroje pro zjišťování podobnosti mezi dvěma soubory v jazyce C. Analyzujte jeho nedostatky.
4. Navrhněte novou verzi tohoto nástroje, která odstraní zjištěné nedostatky.
5. Po konzultaci s vedoucím implementujte nástroj navržený v předchozím bodě.
6. Vytvořené řešení důkladně otestujte sadou minimálně padesáti testů. Zhodnoťte svou práci a diskutujte budoucí vývoj.

Literatura:

- Popis překladače Clang [online]. 2015 [cit. 2015-09-01]. Dostupné na URL: <http://clang.llvm.org/>
- E. Eilam: Reversing: Secrets of Reverse Engineering, Wiley 2005, ISBN 978-076457481.
- Interní dokumentace společnosti AVG.

Pro udělení zápočtu za první semestr je požadováno:

- První čtyři body zadání a rozpracování pátého bodu.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Matula Peter, Ing.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Tato práce se zabývá návrhem, implementací a testováním nástroje *csim*, sloužícího pro porovnávání podobnosti dvou souborů v jazyce C. Primárním účelem vzniku tohoto nástroje je testování zpětného překladače vyvíjeného společností *AVG Technologies s.r.o.* Testování je prováděno na základě podobnosti abstraktního syntaktického stromu původního a dekompilovaného souboru. Čtenář je tedy seznámen se základy problematiky zpětného inženýrství, zejména zpětným překladem binárního kódu do vyšší úrovně reprezentace. Dále je popsán koloběh, kterým kód prochází od jeho vytvoření až po zpětný překlad, a jeho vliv na tento proces. Čtenáři je také poskytnut přehled o projektu LLVM a překladači Clang, který je základním stavebním kamenem nástroje *csim*.

Upozornění: Toto je veřejná zkrácená verze bakalářské práce.

Abstract

This thesis deals with design, implementation and testing of the *csim* tool, which compares two C source files by their similarity. The primary purpose of this tool is testing of a decompiler developed by *AVG Technologies s.r.o.* Testing is based on comparing abstract syntax trees of the original and decompiled source files. The reader is introduced to the basics of reverse engineering, especially reverse engineering of a binary file into a high-level programming language source file. The process of compiling followed by decompiling of a file is described along with its effect on reverse engineering. The LLVM project and the Clang compiler is introduced to the reader, since its libraries are the foundation upon which the *csim* tool is built.

Warning: This is a shortened public version of bachelor thesis.

Klíčová slova

Zpětné inženýrství, zpětný překladač, Clang, LLVM, podobnost zdrojových souborů, podobnost abstraktního syntaktického stromu, detekce malware, jazyk C

Keywords

Reverse engineering, decompiler, Clang, LLVM, source file similarity, abstract syntax tree similarity, malware detection, C language

Citace

REK, Petr. *Detekce podobnosti zdrojových souborů v jazyce C.* Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Matula Peter.

Detekce podobnosti zdrojových souborů v jazyce C

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Petra Matuly.

.....
Petr Rek
15. května 2016

Poděkování

Tímto bych chtěl poděkovat vedoucímu práce Ing. Petru Matulovi za odbornou pomoc a cenné rady při jejím vypracování. Dále bych rád poděkoval vývojářům zpětného překladače *Retargetable Decompiler*, jmenovitě Ing. Jakubu Křoustkovi, za poskytnuté materiály a další rady. Speciální poděkování patří Ing. Petru Zemkovi za velmi užitečné revize zdrojových kódů a implementační tipy.

Obsah

Kapitola 1

Úvod

Zpětné inženýrství je proces extrakce znalostí z čehokoliv vytvořeného člověkem [16]. V této práci nás bude zajímat jeho využití z hlediska softwaru, kterých se nabízí velké množství. Je velmi užitečné v případě, kdy potřebujeme získat hlubší porozumění používaného nástroje nebo při snahách o zdokumentování tzv. *legacy systémů* (zastaralý software) či jejich převodu na nové platformy.

Se schopností analyzovat prakticky jakékoliv existující aplikace však vyvstává mnoho problémů z právního hlediska. Reverzního inženýrství můžeme využít k analýze produktů konkurence, a to jak v kontextu obchodním, tak vojenském. Časté je také využití hackery, kteří se snaží prolomit ochrany produktů, čímž následně dochází k porušování autorských práv, nebo detekovat slabiny systémů či aplikací za účelem jejich zneužití.

A právě naposledy zmíněný fakt je jedním z důvodů, proč jsou antivirové programy v dnešním světě naprostou nutností. Antivirové systémy také hojně využívají reverzního inženýrství, a to jak k analýze škodlivého kódů za účelem ochrany uživatelů, tak pro jeho detekci na základě vzorů. I proto vzniká zpětný překladač *Retargetable Decompiler* [9] vyvíjený společností *AVG Technologies s.r.o.*, ten provádí převod binárního kódu zpět do vyšší úrovně reprezentace (například do jazyka C). Pro jeho testování je nástroj *csim*, který je předmětem této práce, primárně určen.

Nástroj csim slouží k porovnání podobnosti dvou zdrojových souborů v jazyce C. Zaměřuje se zejména na analýzu původního a dekompilovaného kódu. Samotné porovnání je prováděno na úrovni *abstraktního syntaktického stromu* generovaného za pomoci knihoven překladače *Clang* [2]. Kromě testovacích účelů jej lze využít i pro obecné porovnávání dvou souborů a tudíž i detekci malwaru (škodlivého softwaru). Pro toto použití byl nástroj speciálně rozšířen.

Možných využití existuje více. To hlavní se však nachází v použití nástroje během nočních testů zpětného překladače, v těch je nástroj denně používán a odhaluje pokroky či regrese zavedené posledními změnami.

Práce je strukturována následovně. V kapitole 2 je čtenář seznámen se základy zpětného inženýrství, zejména nástroji, kterými zdrojový kód prochází od překladu po zpětný překlad, a jejich vlivem na tento proces. Dále je mu v kapitole 3 přiblížen projekt *LLVM* a pod něj spadající překladač Clang, jehož knihovny jsou základním stavebním kamenem řešení. V kapitole 4 jsou nastíněny problémy, které skýtá porovnání dekompilovaných souborů. V kapitole 5 je rozebráno aktuální řešení a identifikovány jeho klady a zápory. Na základě všech získaných znalostí je poté v kapitole 6 navržen nástroj csim a podpůrná knihovna *csiml*, jejichž implementace je popsána v kapitole 7. Vzhledem k náročnosti testování je tento proces popsán samostatně v kapitole 8. V kapitole 9 je zhodnocena kvalita výsledného řešení a diskutován jeho další vývoj. V závěrečné kapitole 10 je posouzena kvalita této práce.

Kapitola 2

Zpětné inženýrství

Formální definici pojmu reverzní, neboli zpětné, inženýrství v souvislosti se softwarem, která byla poprvé zveřejněna E. J. Chikofskym v časopise *IEEE Software* v roce 1990 [13], lze volně přeložit do následující podoby:

> „Reverzní inženýrství je proces analýzy zkoumaného systému za účelem identifikace jeho komponent a jejich vzájemných vztahů, a vytvoření reprezentace systému v jiné podobě nebo na vyšší úrovni abstrakce."

Tento proces se typicky odehrává s žádnou nebo velmi omezenou znalostí zkoumaného systému.

Reverzní inženýrství zahrnuje širokou škálu technik. Jedním z hlavních způsobů klasifikace je podoba zkoumaného systému, ze které vycházíme. Může se jednat buď přímo o zdrojový kód nebo o spustitelné soubory. Právě zpětný překlad binárních souborů bude centrem zájmu v rámci této práce. Toto odvětví nazýváme *binární reverzní inženýrství* [16].

Pro snazší pochopení problémů, kterým zpětný překlad čelí, budou nyní popsány jednotlivé typy nástrojů, kterými zdrojový kód prochází od jeho vytvoření až po jeho opětovné získání a jejich vliv na tento proces.

2.1 Překladač

Překladač je program, jehož vstupem je kód napsaný ve zdrojovém jazyce a výstupem kód v cílovém jazyce [23]. Zdrojovým jazykem je typicky vysokoúrovňový programovací jazyk, jako například jazyk C. Výstupem pak strojový kód nebo tzv. *bajtkód*, který není závislý na platformě. Zatímco druhé možnosti využívá například jazyk Java [17], nás bude v této práci zajímat první možnost, používaná například jazyky C a C++.

Překlad se obecně sestává z několika fází, u kterých nás, z hlediska zpětného překladu, bude zajímat, které informace jsou v jejich důsledků ztraceny.

2.1.1 Fáze překladu

Preprocesor Před tím, než vůbec můžeme vstupní zdrojový kód přeložit, může být zapotřebí využít programu zvaného *preprocesor*, který například rozgeneruje použitá makra (a tím ztrácíme informaci o jejich použití), odstraní komentáře nebo spojí více zdrojových souborů do jednoho [12].

Lexikální analýza *Lexikální analýza*, první fáze překladu, postupně načítá tzv. *lexémy* zdrojového jazyka, které reprezentuje *tokeny* obsahujícími atributy, jako jsou názvy proměnných, hodnoty číselných konstant a další.

Syntaktická analýza Z tokenů je *syntaktickým analyzátorem* vytvářen derivační strom. Jeho vytváření je řízeno syntaktickými pravidly daného jazyka a zároveň tedy dochází k ověření syntaktické správnosti programu (zápis v jazyce je syntakticky správný, pokud je pro něj možno sestavit derivační strom). V této fázi se ztrácí informace neužitečné pro překlad.

Sémantická analýza Další fází je *sémantická analýza*, která z derivačního stromu vytváří stručnější *abstraktní syntaktický strom*. V průběhu tohoto procesu je ověřována sémantická správnost programu – použité proměnné byly deklarovány, proměnné jsou *ekvivalentní vůči přiřazení* (mají totožný typ nebo je za tímto účelem lze implicitně přetypovat) a další. Sémantická analýza může být spojena se syntaktickou, mluvíme pak o *syntaxí řízeném překladu* [12], v tomto případě je tvorba derivačního stromu simulována a vytvářen je pouze abstraktní syntaktický strom.

Generátor vnitřního kódu Strom z předchozí fáze je vstupem části překladače zvané *generátor vnitřního kódu*, kde dochází k jeho přetváření do interní reprezentace. Ta je překladači vlastní a dokáže s ní lépe pracovat, především ji lépe optimalizovat. Často je využíván *tříadresný kód* (skládající se z kódu instrukce a dvou operandů).

Právě v této fázi dochází ke ztrátě významného množství informací a to v závislosti na úrovni vnitřního kódu. Tyto úrovně existují tři hlavní – nízká, střední a vysoká [19]. Vysokoúrovňové reprezentace jsou bližší zdrojovému kódu a uchovávají více informací, zatímco nízkoúrovňové mají blíže cílovému kódu a spousta informací je ztracena (typy cyklů, přesná struktura podmíněných výrazů, přístupy do pole atd.).

Optimalizátor Vnitřní kód je dále předán *optimalizátoru*, který provádí platformně nezávislé optimalizace. Ty mohou sledovat více hledisek, například velikost výsledného kódu, typicky jsou však prováděny s ohledem na čas vykonání cílového programu. Optimalizací existuje velké množství a to, které budou spuštěny, může uživatel při překladu ovlivnit přepínači (viz sekce 2.1.5). V závislosti na zvolené úrovni optimalizace může docházet k podstatným ztrátám informací, které jsou pro proces zpětného překladu zásadní.

Generátor cílového kódu Generátor cílového kódu spadá pod tzv. *back-end* překladače (předchozím částem se říká *front-end*). V této části je optimalizovaný vnitřní kód přetvářen na cílový kód – tím je typicky jazyk symbolických adres nebo binární kód. Mimo to může také docházet k provádění dalších, platformně nebo jazykově specifických, optimalizací.

Jak je patrné, největší problém z pohledu zpětného inženýrství představují dvě části překladače – generátor vnitřního kódu a optimalizátor.

2.1.2 Symbolická jména

Symbolickými jmény máme na mysli názvy funkcí a globálních proměnných. Ty jsou většinou při překladu zachovány, a jak později uvidíme, porovnávání souborů se zachovanými symbolickými jmény je značně usnadněno.

V některých případech však může dojít k jejich ztrátě, především snahou tvůrce kódu a to právě za účelem znesnadnění procesu zpětného překladu. K tomuto účelu můžeme využít speciálních programů (které zvládají odstranit i ladicí informace, jako třeba *strip*), ale tuto činnost zvládá i překladač (přepínač `-s` u GCC či Clang).

2.1.3 Datové typy

Při převodu optimalizovaného vnitřního kódu do cílového kódu dochází ke ztrátě informací o datových typech, neboť jak na úrovni binárního kódu, tak v jazyce symbolických adres, existují pouze dva základní – s pevnou řádovou čárkou a plovoucí řádovou čárkou. Ty lze dále dělit podle jejich *šířky* (počet bitů), i tak však nutně nemusí odpovídat typu, který byl programátorem původně zvolen.

Zpětná inference datových typů je problematická a patří mezi jednu z největších výzev zpětného překladu. S jistotou můžeme rozeznat pouze typy těch proměnných, které byly použity v rozeznaných knihovních funkcích vyžadujících daný typ parametru [15]. Případně můžeme získat podrobnější informace o typu na základě použitých instrukcí (například porovnání znaménkových a bezznaménkových typů se na úrovni instrukcí liší).

Objektivní porovnání podobnosti datových typů ve zdrojových souborech je tedy problematické, neboť odlišnosti nemusí být nutně problémem zpětného překladače.

2.1.4 Ladicí informace

Při překladu můžeme (většinou přepínačem `-g`) aktivovat vkládání ladicích informací, určených pro programy zvané *debugger* (ladicí nástroj, viz sekce 2.4). Ty sice zvyšují velikost výsledného binárního souboru, ale slouží k jednoduššímu ladění kódu či hledání chyb. Ze své podstaty musí obsahovat přesné informace o struktuře kódu (aby například mohly uživateli poskytnout informaci o tom, na jakém řádku či v jaké funkci k chybě došlo). Uchovávanou informací je to, které konstrukce zdrojového jazyka odpovídají části kódu v jazyce cílovém.

Zpětný překlad binárního souboru obsahujícího tyto informace je zjednodušen a dosažené výsledky podobnosti se značně liší v závislosti na jejich dostupnosti.

2.1.5 Optimalizace

Prováděných optimalizací existuje větší množství a uživatel většinou může ovlivnit, které z nich budou spouštěny vybráním úrovně optimalizace. Výchozím úkolem překladače je redukovat časovou náročnost překladu a dohlédnout na to, že kód bude snadno laditelný (optimalizace ovlivňují strukturu kódu a ladění by pak nemuselo poskytovat korektní informace).

V překladači GCC (a také Clang) je tato defaultní úroveň označena přepínačem `-O0` [5]. Přepínačem `-O` či `-O1` instruujeme překladač, aby se pokusil redukovat velikost kódu a čas vykonání. I přesto, že časově náročné optimalizace nejsou využity, tak čas překladu mírně stoupá. Existuje také přepínač `-O2` a `-O3`. Větší číslo znamená efektivnější kód, z hlediska časové i paměťové náročnosti, ale delší čas překladu. Zároveň také dochází k čím dál rozsáhlejším úpravám zdrojového kódu v souvislosti s prováděnými optimalizacemi a podobnost vůči původnímu kódu klesá.

Dále je popsáno několik základních optimalizací, z nichž některé představují velký problém pro analýzu podobnosti i zpětný překladač samotný.

Eliminace mrtvého kódu Jedná se o jednu z nejčastějších optimalizací, která je z hlediska velikosti výsledného programu velmi podstatná. *Mrtvým kódem* se myslí takový kód, který není nikdy vykonán (funkce není volána, podmínka je vždy nepravdivá atd.) a jeho přítomnost v cílovém kódu je tedy zbytečná.

Z hlediska zpětného překladu, oproti jiným optimalizacím, představuje spíše výhodu, neboť snižuje množství kódu, který je třeba analyzovat, aniž by se to jakkoliv podepsalo na funkčnosti. V případě nástroje pro porovnání dvou zdrojových souborů však činí problém, neboť důsledkem je snížení podobnosti o chybějící kód, za což zpětný překladač nemůže.

Inlining *Inlining* je proces dosazení těla funkce na místo jejího volání. To se provádí, pokud je tělo funkce primitivní (například jednořádkové) a režie spojená s voláním funkce by byla neúměrně vysoká, i na vyžádání programátorem (klíčovým slovem `inline` v jazyce C – rozhodnutí o provedení je však stále na překladači). Názorně je efekt této optimalizace znázorněn v ukázkách níže, zde si můžeme všimnout, že tělo funkce `function_B()` bylo vloženo na místo jejího volání ve funkci `function_A()` (ukázka 2.1), výsledkem je funkce `function_A_B()` (ukázka 2.2).

Inlining představuje z hlediska analýzy podobnosti velký problém, neboť mění podobu grafu volání, který je pro porovnání, především správné promítnutí funkcí jednoho souboru na funkce v druhém, zásadní.

```
int function_A(int *arr, int n) {
        for (int i = 0; i < n; i++) {
                arr[i] = function_B(arr[i], 5);
        }
        printf("Values␣to␣the␣fifth␣power.");
}

static inline int function_B(int a, int b) {
        int retval = a;
        for (int i = 0; i < (b-1); i++) {
                retval *= a;
        }
        return retval;
}
```

Ukázka 2.1: Stav před provedením inliningu

```
int function_A_B(int *arr, int n) {
        for (int i = 0; i < n; i++) {
                int a = arr[i];
                for (int b = 0; b < 4; b++) {
                        arr[i] *= a;
                }
        }
        printf("Values␣to␣the␣fifth␣power.");
}
```

Ukázka 2.2: Stav po provedení inliningu

Zjednodušování matematických výrazů Matematické operace jsou v programech velmi časté a proto také existuje velké množství jejich optimalizací. Mezi ty základní patří dosazování do výrazu pro snížení počtu proměnných, výpočet statických matematických výrazů nebo jejich úpravy na ekvivalentní, ale efektivnější (typickým příkladem je převedení dělení či násobení na bitové posuvy).

Zatímco zjednodušování představuje výzvu pro tvůrce zpětného překladače (je vhodnější převést zjednodušený výraz do původní podoby snáze pochopitelné programátorem), nástroj pro analýzu podobnosti to neovlivní, neboť porovnávání dvou souborů je typicky prováděno na vyšší úrovni, než jsou jednotlivé výrazy.

Reorganizace kódu ve smyčkách V případě, že je smyčka vykonána pouze několikrát, je možné její tělo do výsledné podoby vložit několikrát za sebe. Tím sice zvýšíme velikost programu, avšak ušetříme čas spojený s podmíněnými skoky. V rámci smyček je také časté přeuspořádání instrukcí a vyřazení těch, které jsou časově nezávislé, před tělo smyčky (takzvaný *hoisting*).

Eliminace jednotlivých smyček problémem pro analýzu podobnosti je, neboť ty lze jednoznačně k porovnávání využít.

2.2 Assembler a disassembler

Assembler slouží k převodu kódu v platformně specifickém jazyce symbolických instrukcí (taktéž zvaném jazyk symbolických adres nebo nepřesně assembler) do binární podoby, kterou procesor zvládne vykonat. Tato činnost není příliš složitá, ale může se komplikovat prováděnými optimalizacemi.

Úkolem *disassembleru* je pak převod opačným směrem. Tento proces již tak jednoduchý není. Problematické je například rozeznání dat od instrukcí ve *Von Neumannově architektuře* (obsahuje společnou paměť pro data i instrukce), tento problém se poté násobí v případě, že jsou využívány instrukce s proměnnou délkou. Jen jeden jediný špatně rozeznaný bajt nebo špatně rozeznaná instrukce mohou způsobit velké problémy ve zbytku analyzovaného binárního kódu – v případě, že místo instrukce o délce dva bajty rozeznáme instrukci o délce tři bajty, všechny další instrukce jsou taktéž posunuty a tudíž chybně rozeznány.

Kód v jazyce symbolických instrukcí může posloužit k pochopení činnosti aplikace, avšak vyžaduje více úsilí, znalostí a je také časově náročnější než zkoumání kódu ve vysokoúrovňové reprezentaci. Také úprava takto získaného kódu může být problematická, například kvůli převodům mezi absolutními a relativními skoky, které assembler provádí. Proto vznikají složitější nástroje zvané *zpětný překladač*.

2.3 Zpětný překladač

Zpětný překladač se značně liší od disassembleru. Jeho výstup je kratší a lépe srozumitelný. Zatímco zpětný překlad bajtkódu je již do jisté míry zvládnutý [18], neboť je zachováno mnohem větší množství informací o typech či struktuře kódu, proces dekompilace strojového kódu je stále velmi složitým problémem vzhledem k velkému množství chybějících informací. Právě zpětný překlad binárního programu nás bude dále zajímat.

> „Dekompilátor, neboli zpětný překladač, je program snažící se provést opak překladu. Cílem je vytvoření funkcionálně ekvivalentního programu ve vysokoúrovňovém jazyce z vstupního binárního programu. Vstup je platformně specifický, zatímco výstup jazykově specifický.“ [14]

Důvodů, proč zpětný překladač využívat, existuje více. Může být použit například při snaze o získání ztracených zdrojových kódů nebo pro převod starších programů na nové platformy. Zpětný překladač společnosti AVG se zaměřuje zejména na jeho využití pro detekci škodlivého softwaru.

2.3.1 Obecná struktura zpětného překladače

Zpětný překladač je možno rozdělit na tři hlavní části – strojově specifický *front-end* [21], nezávislý *middle-end* a jazykově specifický *back-end* [14]. Jazykově i strojově nezávislý middle-end je podstatný zejména proto, že umožňuje převádět do vnitřní reprezentace libovolný podporovaný kód a produkovat z něj kód v různých vysokoúrovňových jazycích.

Front-end Cílem tohoto modulu je načíst strojově specifický kód do vnitřní reprezentace. Mohli bychom jej dále rozdělit na *loader* (někdy také řazen do části zvané *preprocessing* [21]), *parser* a *sémantickou analýzu*.

Loader zjistí základní informace o spustitelném souboru, jako například architekturu, dostupnost ladicích informací, použití packeru či vstupní bod programu, a strojový kód načte do paměti. Parser strojový kód převádí do interní reprezentace, kterou strukturuje. Tím vznikají tzv. *základní bloky* (anglicky *basic block* [16]), reprezentující tok programu (cykly, funkce atd.). Jeden základní blok odpovídá několika málo příkazům ve vysokoúrovňovém jazyce. Sémantická analýza tuto reprezentaci dále upravuje, podstatná je zde zejména inference datových typů a rozeznání známých *idiomů* (způsobů, kterými jsou přeloženy určité vysokoúrovňové příkazy), které je možno převést do srozumitelnější podoby.

Middle-end Vnitřní reprezentace má typicky dvě úrovně, nižší, která je podobná jazyku symbolických instrukcí a vyšší, která se podobá vysokoúrovňovému jazyku. Tato část provádí převod mezi těmito dvěma úrovněmi, k čemuž opět využívá detekce idiomů nebo si pomáhá analýzou toku dat.

Podstatná je také detekce funkcí, které nebyly vytvořené uživatelem, ale pochází ze známých knihoven. Tato detekce je prováděna pomocí signatur a napomáhá čitelnosti výsledného kódu. Mimo knihovních funkcí je také nutno odstranit inicializační funkce. I v prostém programu typu *Hello, world!* (viz ukázka 2.3) s jednou funkcí ve zdrojovém kódu se může vyskytovat až 23 funkcí v kódu cílovém [14].

Výstupem této fáze je kód strukturovaný do bloků a funkcí, obsahující datové typy, včetně těch uživatelsky definovaných (struktury), který je uživatelem lépe čitelný neboť došlo také k navrácení různých optimalizací (bitové posuvy, cykly s podmínkou na konci).

Back-end Poslední část zpětného překladače je jazykově specifická, avšak umožňuje převádět interní reprezentaci do různých podporovaných jazyků. Pro převod využívá vytvořeného grafu volání a grafu toku řízení. Vnitřní kód je převáděn na ekvivalentní výrazy v daném jazyce. To do jaké míry jsou podporovány jazykově specifické konstrukce pak určují samotní vývojáři.

2.4 Debugger

Debugger (ladicí nástroj) je jedním z velmi užitečných nástrojů při zpětném inženýrství prováděném manuálně uživatelem. Ten spustí binární soubor a sleduje jeho běh, může tak analyzovat tok řízení a podstatné části, na které by se měl dále detailněji zaměřit.

Mimo to ladicí nástroj poskytuje širokou škálu možností pro dynamickou analýzu zdrojového kódu, které jsou velmi užitečné, zejména pokud jsou integrovány do vývojového prostředí (break pointy, zjištění obsahu paměťových buněk za běhu a další).

V případě dekompilace prováděné automaticky nemá debugger prakticky žádný význam, avšak informace pro něj určené mohou být velmi užitečné pro zpětný překladač, neboť poskytují detailní informace o původní struktuře zdrojového kódu.

2.5 Obrana proti zpětnému inženýrství

Je logické, že reverzní inženýrství není vždy žádoucí a například tvůrci proprietárního softwaru se snaží tomuto procesu zabránit nebo jej alespoň ztížit. Zatím neexistuje absolutní ochrana, avšak je možno tento proces zkomplikovat na takovou úroveň, že není časově ani ekonomicky zvládnutelný. Každá takováto snaha má ale také svou cenu, která se typicky projeví na velikosti nebo výkonu chráněné aplikace (zvláště v případě obfuskace kódu).

Základním prvkem obrany proti procesu reverzního inženýrství je mazání symbolických informací, mezi které patří názvy funkcí nebo globálních proměnných – tuto jednoduchou činnost typicky zvládá i překladač. Pokročilé postupy pak zahrnují nástroje popsané dále.

2.5.1 Obfuskátor kódu

Jedná se o nástroj, který se snaží snížit čitelnost kódu různými změnami jeho struktury, včetně přidávání neužitečných částí či skrývání řetězců. Výsledná podoba má stejnou funkcionalitu, avšak je mnohem hůře čitelná programátory.

Existují nástroje zvané *deobfuskátory*, které jsou výhodné zejména pro eliminaci neužitečného kódu. Na základě analýzy toku programu dokáží detekovat slepé uličky a tím pádem také kód, který nemá žádný reálný dopad na funkcionalitu systému. Zvládají i strukturovat kód do čitelnější podoby, tato činnost je však mnohem komplikovanější.

2.5.2 Packer

Packer je typ aplikace, která komprimuje nebo šifruje obsah spustitelného souboru. Existují dva typy těchto nástrojů

První z nich, využívající komprimaci, slouží k pouhému snížení velikosti binárního souboru. Při spuštění je poté uplatněn algoritmus, který obsah dekomprimuje. Tento proces není příliš složitý a je jednoduché jej zvrátit.

Druhý typ navíc využívá šifrování, jehož cílem je znesnadnit proces reverzního inženýrství. Aplikace je zašifrována a poté dešifrována až za běhu na základě klíče. Představuje to komplikaci pro statickou analýzu kódu, analýza kódu za běhu nijak netrpí. Získání zdrojového kódu může být složitější, neboť pokročilé nástroje (jako například Themida [10]), postupně dešifrují pouze aktuálně využívanou část aplikace. Ta tady v paměti není v žádný okamžik celá. V některých případech je však možno aplikaci dešifrovat a získat kód i pro statickou analýzu.

K rozšifrování a dekomprimaci slouží nástroje zvané *unpacker*.

2.5.3 Antidebugger

Součástí chráněné aplikace je speciální kód, který se snaží zamezit funkci ladicích nástrojů, případně detekovat jejich přítomnost a ukončit se [22]. Debuggery jsou totiž velmi často využívány při manuální dekompilaci, zejména pro získání dešifrovacího klíče.

Tyto snahy jsou však typicky platformně specifické, nepříliš spolehlivé a je možné je obejít.

2.6 Zpětný překladač společnosti AVG

Dekompilátor, který je středem zájmu v této práci, nese název *Retargetable Decompiler* [9]. Jedná se o univerzální nástroj, který není závislý na architektuře počítače ani formátu binárního souboru (a v oblasti rekonfigurovatelných zpětných překladačů ani neexistuje konkurence [20]). Využívá podobné struktury, jaká byla popsána v sekci 2.3.1. Během vývoje nástroje csim podporoval následující architektury – *x86* (respektive *IA-32*, neboť je podporována pouze 32 bitová verze, to platí i pro další zmíněné), *ARM*, *MIPS*, *PIC32* a *PowerPC*.

Vstupem je binární soubor, ve formátech *PE*, *ELF* či *COFF*, a výstupem kód v jazyce C nebo v jazyce podobném Pythonu. Kromě toho ale také dokáže generovat graf volání, výstup v jazyce symbolických instrukcí, graf toku řízení a další statistiky.

Jeho velkým plusem je schopnost detekovat knihovní funkce na základě jejich otisků, což výrazně zvyšuje čitelnost kódu. Dokáže také dobře napodobit původní strukturu kódu, tedy funkce, cykly nebo struktury. K tomu využívá také ladicí informace jak ve formátu *PDB*, tak i *DWARF* [21].

Součástí vyvíjeného zpětného překladače jsou i podpůrné nástroje, jako například nástroj pro dešifrování či extrahování binárního souboru (unpacker). Pro testování kvality zpětného překladu byl původně využíván nástroj *csimilarityCMP*, jehož náhradou je nástroj *csim*, vyvinutý v rámci této práce.

Dekompilaci si může čtenář vyzkoušet sám na webu [9] a s pomocí webového *API* (aplikační rozhraní) také integrovat zpětný překlad do svého vlastního nástroje. Zde je k dispozici krátká ukázka kódu před překladem, ukázka 2.3, a po zpětném překladu, ukázka 2.4. Složitější ukázky pak budou poskytnuty dále v kapitole 4.

```
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

Ukázka 2.3: Program *Hello, world!* před překladem

```
#include <stdint.h>
#include <stdio.h>

// Address range: 0x407740 - 0x40775f
int main(int argc, char ** argv) {
    ___main(); // 0x407740
    puts("Hello, world!");
    return 0;
}
```

Ukázka 2.4: Program *Hello, world!* po zpětném překladu

Kapitola 3

Projekt LLVM

LLVM [8] je rozsáhlým aktivně vyvíjeným open-source projektem, který původně vznikl jakožto výzkumná práce na univerzitě v americké Illinois v roce 2000. Zahrnuje překladač *Clang* [2] a sadu nástrojů užitečných při psaní aplikací pro statickou analýzu nebo provádění optimalizací.

Mimo již zmíněné zahrnuje projekt LLVM i další nástroje a knihovny, jako je například debugger *LLDB*, *libc++*, což jsou optimalizované standardní knihovny jazyka C++ nebo *LLVM Core* provádějící jazykově nezávislé optimalizace nad *LLVM IR* (viz sekce 3.1).

3.1 LLVM IR

LLVM IR (*LLVM Intermediate Representation*) je interní reprezentace využívána překladačem Clang a dalšími nástroji (například i zpětným překladačem Retargetable Decompiler [20]). Reprezentace je podobná jazyku symbolických instrukcí, avšak abstrahuje od jakýchkoliv strojově specifických instrukcí. Namísto registrů také využívá proměnné.

I když je primárně určen pro použití při překladu programů z jazyků C a C++, je možno za pomoci knihoven LLVM převést do interní reprezentace celou řadu dalších jazyků, jako například Ruby nebo Python [6].

Celkem jsou poskytovány tři úrovně reprezentace – formát podobný jazyku symbolických instrukcí (viz ukázka 3.1), který je dobře čitelný, formát vhodný pro zpracování frontendovými nástroji a bajtkód, určený především pro serializaci.

```
@.str = private constant [13 x i8] c"Hello, world!\00"
declare i32 @printf(i8*, ...)

; Function Attrs: nounwind uwtable
define i32 @main(i32 %argc, i8** \%argv) {
  %1 = alloca i32
  %2 = alloca i32
  %3 = alloca i8**
  store i32 0, i32* %1
  store i32 %argc, i32* %2
  store i8** %argv, i8*** %3
  %4 = call i32 (i8*, ...)* @printf(i8*
        getelementptr inbounds ([13 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}
```

Ukázka 3.1: Část LLVM IR odpovídající programu *Hello, world!* (viz ukázka 2.3)

3.2 Clang

Clang je především znám jako překladač jazyků C, C++ a Objective C, který jako svou interní reprezentaci využívá právě LLVM IR. V počátku byl vyvíjen zejména jako součást překladače *GCC* optimalizovaná pro překlad Objective C, avšak brzo se projekt transformoval v samostatný překladač.

3.2.1 Clang AST

Clang AST je reprezentace zdrojového kódu v podobě abstraktního syntaktického stromu (anglicky ***A**bstract **S**yntax **T**ree*), jehož struktura je velmi podobná C++ kódu a tudíž i dobře analyzovatelná.

Abstraktní syntaktický strom je pro činnost překladače velmi podstatný a je typicky výsledkem jeho syntaktické nebo sémantické analýzy. Reprezentuje zdrojový kód, kde každý uzel stromu představuje jednu určitou konstrukci daného jazyka. V této reprezentaci může být kód sémanticky i syntakticky ověřen, převeden do interní reprezentace a dále optimalizován.

Tento strom je díky kvalitnímu API možno efektivně procházet. Je složen z uzlů. Těmi základními jsou:

- `Type` – reprezentující datové typy,
- `Decl` – reprezentující veškeré deklarace, které jsou jeho podtřídami, mezi ty patří deklarace proměnných (`VarDecl`) či funkcí (`FuncDecl`),
- `Stmt` – reprezentující všechny příkazy a výrazy, dále je upřesňován podtřídami jako `Expr` (matematický výraz), `ForStmt` (cyklus `for`) nebo `ReturnStmt` (návrat z funkce).

```
cdefs.h:55:54> /usr/include/stdio.h:920:13 funlockfile 'void (FILE *)' extern
| |-ParmVarDecl 0x3a3d290 <col:26, col:32> col:32 __stream 'FILE *'
| `-NoThrowAttr 0x3a3d3c0 </usr/include/x86_64-linux-gnu/sys/cdefs.h:55:35>
`-FunctionDecl 0x3a3d550 <hello.c:3:1, line:6:1> line:3:5 main 'int (int, char **)'
  |-ParmVarDecl 0x3a3d410 <col:10, col:14> col:14 argc 'int'
  |-ParmVarDecl 0x3a3d480 <col:20, col:27> col:27 argv 'char **'
  `-CompoundStmt 0x3a3d780 <col:33, line:6:1>
    |-CallExpr 0x3a3d6e0 <line:4:2, col:23> 'int'
    | |-ImplicitCastExpr 0x3a3d6c8 <col:2> 'int (*)(const char *, .)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x3a3d600 <col:2> 'int (const char *)' Function 0x3a2f220 'printf' 'int (const char *)'
    | `-ImplicitCastExpr 0x3a3d728 <col:9> 'const char *' <BitCast>
    |   `-ImplicitCastExpr 0x3a3d710 <col:9> 'char *' <ArrayToPointerDecay>
    |     `-StringLiteral 0x3a3d668 <col:9> 'char [13]' lvalue "Hello, world!"
    `-ReturnStmt 0x3a3d760 <line:5:2, col:9>
      `-IntegerLiteral 0x3a3d740 <col:9> 'int' 0
```

Ukázka 3.2: Výsek *Clang AST* odpovídající programu *Hello, world!* (viz ukázka 2.3)

3.2.2 Rozhraní Clang

Clang, jakožto knihovna, poskytuje rozhraní [1] umožňující psát nástroje vyžadující syntaktické či sémantické informace o zdrojovém kódu. Celkem existují tři dostupná rozhraní:

- LibClang,
- Clang Plugins,
- LibTooling.

LibClang *LibClang* je rozhraní napsané v C, ale využitelné i v C++ či Pythonu (za pomocí *C bindings*). Rozhraní je spíše stabilní a snaží se zachovávat zpětnou kompatibilitu napříč novými verzemi.

Poskytuje přístup k abstraktnímu syntaktickému stromu na vysoké úrovni s velkým množstvím abstrakcí. Není tudíž vhodné v případě, že chceme mít plnou kontrolu nad Clang AST.

Clang Plugins Rozhraní *Clang Plugins* umožňující úpravy a analýzu abstraktního syntaktického stromu během překladu a to v podobě přídavných akcí, které jsou vykonány *front-endem* překladače.

LibTooling Rozhraní *LibTooling* je vytvořené v C++ a poskytuje úplnou kontrolu nad AST. Lze jej využít k psaní nástrojů nezávislých na překladači.

Problémem je, že se nehledí na zachovávání zpětné kompatibility a rozhraní se napříč novými verzemi často mění. Rozhraní také neposkytuje prakticky žádnou abstrakci od implementace a vyžaduje tedy jeho větší znalost programátorem.

3.2.3 Clang versus GCC

Výhodou GCC, kterou se však projekt LLVM snaží eliminovat, je širší podpora jazyků a také cílových platforem.

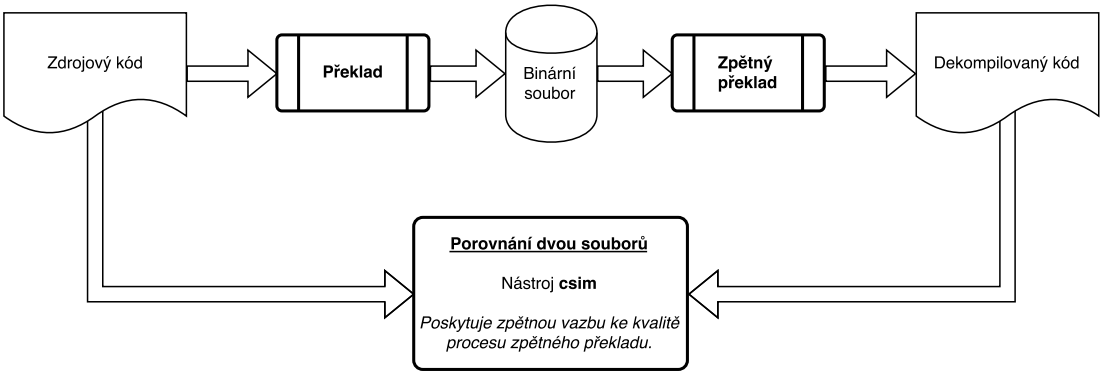
Naopak výhod Clangu je celá řada [3]. Clang a jím nabízené knihovny, zejména v souvislosti s abstraktním syntaktickým stromem, jsou velmi dobře použitelné a čitelné. Většina nástrojů navíc nabízí veřejná a dokumentovaná API, umožňující statickou analýzu zdrojových kódů. Tato výhoda je patrná zejména v nástroji csim. Clang také nabízí snáze pochopitelná a detailnější chybová hlášení.

Co se týče rychlosti, existují oblasti, kde Clang vyniká, ale i oblasti, kde GCC stále dominuje [7]. Vývojáři překladače Clang však tuto mezeru s každým novým vydáním minimalizují.

Kapitola 4

Problematika porovnání dekompilovaných souborů

Cílem této bakalářské práce (viz obrázek 4.1) je vytvořit nástroj, který porovnává dva soubory – původní a dekompilovaný. Výstupem takového porovnání je nejen celková podobnost, ale i podobnost dle různých kritérií. Ty poskytují vývojářům zpětnou vazbu k provedeným změnám a informují o oblastech, v nichž zpětný překladač vyniká a ve kterých nikoliv.



Obrázek 4.1: Znázornění cíle této bakalářské práce

Porovnání zahrnující dekompilované soubory má svá specifika, která značně komplikují tento proces. Zatímco existují nástroje pro porovnání dvou obecných zdrojových kódů nebo dvou binárních souborů, autorovi se nepodařilo nalézt žádný nástroj snažící se o podobné porovnání, o které se snaží nástroj csim. A to ani nástroj, který by obdobným způsobem porovnával abstraktní syntaktické stromy obecných zdrojových souborů.

Samozřejmě šlo nalézt nástroje, které porovnávají dva soubory. Jedním z nich byl *BinDiff* [11], který porovnává binární soubory. Ten bude také detailněji popsán dále, neboť byl využíván jako referenční. Nástrojů pro porovnání zdrojových kódů na úrovni textu existuje více, můžeme zmínit například webový *DiffNow* [4], avšak porovnání je opravdu prováděno pouze na textové úrovni – chybějící nebo přebývající slova či řádky atd. I když DiffNow umožňuje specifikovat porovnání zdrojových kódů jazyka C/C++ a poté nabízí různá vylepšení (například uvažuje `return EXIT_FAILURE` za ekvivalentní k `return 1`), tak je takovéto porovnání v případě originálního a dekompilovaného souboru naprosto nevhodné. I proto se dále budeme bavit spíše o hypotetických nástrojích, porovnávajících abstraktní syntaktické stromy dvou obecných kódů a rozdílech, při porovnání původního a dekompilovaného kódu.

Také fakt, že získání izomorfního kódu (totožná struktura původního a dekompilovaného kódu) není ze své podstaty možné, značně ovlivňuje proces porovnání těchto souborů. Maximální podobnost u klasických nástrojů znamená, že jsou soubory totožné – jejich struktura, parametry funkcí, graf volání a další. Maximální podobnost u nástroje csim však spíše vyjadřuje, že zpětný překladač vynaložil maximální úsilí pří získávání původní podoby kódu vzhledem k informacím, které měl k dispozici. Z toho logicky vyplývá, že musí tolerovat různé nedostatky, které zpětný překladač nemohl ovlivnit.

4.1 Porovnání dekompilovaného souboru s původním

Soubory lze na úrovni zdrojových kódů porovnávat z více hledisek. Tím stěžejním, a také nejkomplikovanějším, je porovnání funkcí. Jádrem tohoto porovnání je proces promítnutí funkcí jednoho souboru na jejich protějšky v druhém souboru. Pokud jsou dostupné symboly (a je známo, že nedošlo k přejmenování funkcí), můžeme je k promítnutí využít. Tím je celý proces značně zjednodušen. Pokud se na ně však nemůžeme spolehnout, je nutno provést promítnutí na základě podobnosti atributů funkcí.

Atributů je mnoho a porovnávaná množina se může napříč různými nástroji lišit. Ty pro porovnání dvou obecných zdrojových kódů by například mohly porovnávat typy parametrů a proměnných, použité příkazy, typy cyklů a další – na žádnou z těchto informací se nelze při porovnávání dekompilovaného souboru s původním spolehnout.

Nástroje pro porovnání binárních souborů využívají typy instrukcí či délky těl funkcí – což se překladem a následnou dekompilací taktéž silně mění. Inference datových typů po překladu je velmi složitá a původní podoby ani nelze dosáhnout. Typy cyklů jsou překladači často měněny, to samé platí i pro strukturu podmíněných výrazů. Délka funkcí se pak často liší i více než desetinásobně.

Velmi slibně může znít idea promítání funkcí skrze *graf volání* [1]. I ten však nemusí být zachován, zejména kvůli již zmíněné optimalizaci zvané *inlining*. A to už vůbec nebereme v potaz možné chyby zpětného překladače, kdy by i jedna nedetekovaná funkce zmařila výsledky celé analýzy. Přesto je to jeden z nejspolehlivějších ukazatelů, na kterém lze řešení jednoznačně založit, jak bude ukázáno později.

Různých již zmíněných nedostatků, které musí nástroj csim tolerovat, bylo detekováno větší množství. Při návrhu nástroje csim a následné implementaci na ně byl brán zřetel. Níže můžeme nalézt krátký výčet některých těchto problémů.

- inference celočíselných datových typů na typy s definovanou šířkou
  `int` ⇒ `int32_t`
  - s tím souvisí vkládání hlavičkového souboru `stdint.h` definujícího tyto typy
- záměna proměnných za ukazatele na proměnnou
  `int` ⇒ `int *`
- náhrada návratových typů `void` za `int`
  `void function()` ⇒ `int32_t function()`
- neschopnost detekovat typy cyklů
- neschopnost rekonstruovat strukturu složených podmíněných příkazů

[1] Graf volání je orientovaný graf, jehož každý uzel představuje funkci a každá hrana popisuje vztah dvou funkcí, kde jedna stojí v pozici volajícího a druhá v pozici volaného.

- rozdíly v grafech volání jinak funkčně ekvivalentních programů

Problematika se dále komplikuje s rostoucí počtem prováděných optimalizací. Nástroj musí být schopen porovnat i soubory, které spolu na první pohled nemají nic společného a objektivně vyjádřit jejich podobnost.

4.2 Porovnání dvou dekompilovaných souborů

I když se práce primárně zaměřuje na porovnání původního programu s jeho dekompilovanou variantou, je řešení rozšířeno i na porovnání dvou dekompilovaných souborů. Hlavní využití se, vzhledem k zaměření zpětného překladače, nalézá v porovnání dvou dekompilovaných vzorků, z nichž jeden je škodlivým softwarem (malwarem). Samotná podobnost poté představuje pravděpodobnost, že druhý soubor je taktéž stejnou rodinou malwaru. Podobně bychom mohli tento nástroj využít i pro porovnání dvou vzorků virů (s časovým rozestupem) a detekci změn, které byly provedeny.

Protože před procesem zpětného překladu neexistuje stoprocentní ochrana, zaměřují se tvůrci malware spíše prodloužení doby, po kterou je nově vyvinutý virus, nebo jeho pozměněná verze, schopen unikat detekci antivirovými systémy [16]. Čím delší tato doba je, tím více počítačů je infikováno. Detekce virů je typicky prováděna porovnáním otisku souboru s databází otisků virů. Cílem je tedy změnit otisk nové verze viru natolik, že se již nepodobá svému předchůdci. K tomuto je možno využít například následujících dvou postupů – *polymorfismu* a *metamorfismu* [22].

Polymorfismus *Polymorfismus* je proces šifrování zdrojového kódu, který je poté opětovně dešifrován za běhu, a vkládání neužitečných částí před a za zdrojový kód. Hlavním nedostatkem této metody je fakt, že samotný šifrovací algoritmus (který na nejvyšší úrovni logicky šifrovaný není a ani být nemůže) lze využít jako otisk k detekci viru.

Metamorfismus *Metamorfismus* je složitější a mnohem více nebezpečný, neboť dokáže déle unikat odhalení. Kód při *replikaci* (infikování dalšího počítače) sám mění svou strukturu – to zahrnuje spojování či rozdělování funkcí, vkládání neužitečného kódu, prohazování pořadí instrukcí nebo celých funkcí, převod podmínek na ekvivalentní a další menší či větší změny. Této metody využívá mnoho rozšířených malwarů a takto upravené viry je obtížnější detekovat.

Nástroj csim, který je psán speciálně za účelem objektivního porovnání dvou souborů, které se mohou značně lišit, zde může dosahovat zajímavých výsledků, zejména v módu porovnání dvou dekompilovaných souborů (kde lze využít optimalizací jinak nepoužitelných při porovnání původního a dekompilovaného souboru).

4.3 Ukázkové porovnání

Aby si čtenář lépe dokázal představit zmíněné problémy, je přiložena následující ukázka zobrazující podobu dekompilovaného kódu a jeho vztah k původnímu, který se, jak již bylo zmíněno, značně liší se stoupající úrovní optimalizací.

Zdrojový kód byl přeložen překladačem *GCC* na operačním systému *Windows* pro architekturu *Intel x86*. Nebylo využito vkládání ladicích informací a došlo k odstranění symbolických jmen (názvy, které vidíte v ukázce 4.2 a v ukázce 4.3 byly vytvořeny zpětným překladačem). Spouštěné optimalizace se liší napříč ukázkami.

4.3.1 Původní kód

Zdrojovým kódem je poněkud komplikovaný program typu „Hello, world!“ (ukázka 4.1) obsahující funkce, struktury a cykly. Ten je přeložen a poté dekompilován zpětným překladačem Retargetable Decompiler verze `2.1.2` zveřejněné 27. ledna 2016 (v ukázkách chybí komentáře, obsahující například adresový rozsah funkce, z důvodu snadnější čitelnosti a vyšší kompaktnosti).

```
#include <stdio.h>
#include <stdlib.h>

struct Delimiter{
    char character;
};

char* string[] = {"Hello,", "world!"};

void print(int offset) {
    printf(string[offset]);
}

int main(int argc, char **argv) {
    struct Delimiter *delChar = malloc(sizeof(struct Delimiter));
    delChar->character = '␣';

    for (int i = 0; i < 2; i++) {
        print(i);
        printf("%c", delChar->character);
    }
    printf("\n");

    return 0;
}
```

Ukázka 4.1: Zdrojový kód složitějšího programu typu *Hello, world!*

4.3.2 Dekompilovaný kód

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int32_t function_401560(int32_t a1);

char * g1[2] = {
    "Hello,",
    "world!"
};


int32_t function_401560(int32_t a1) {
    return printf(g1[a1]);
}
```

```
int main(int argc, char ** argv) {
    ___main();
    char * mem = malloc(1);
    *mem = 32;
    for (int32_t i = 0; i < 2; i++) {
        function_401560(i);
        putchar((int32_t)*mem);
    }
    putchar(10);
    return 0;
}
```

Ukázka 4.2: Dekompilovaný kód, bez optimalizací při překladu

V předcházející ukázce 4.2 nebylo využito žádných přepínačů optimalizace (ekvivalentní přepínači `O0`). Lze vidět, že kódy jsou si strukturálně velmi podobné. I přesto si však můžeme všimnout některých problémů zmíněných dříve – struktura `Delimiter` úplně chybí, datový typ `int` byl nahrazen za `char*`, návratový typ funkce `print` již není `void`, ale `int32_t`. V hlavní funkci `main` došlo k přidání proměnné a volání `printf(\n)` bylo nahrazeno `putchar(10)`, kde 10 je ASCII kód znaku nového řádku.

```
#include <stdint.h>
#include <stdio.h>

int main(int argc, char ** argv) {
    ___main();
    printf("Hello,");
    putchar(32);
    printf("world!");
    putchar(32);
    putchar(10);
    return 0;
}
```

Ukázka 4.3: Dekompilovaný kód, s nejvyšší úrovní optimalizací při překladu

V poslední ukázce 4.3 bylo využito maximální optimalizační úrovně `O3`. Můžeme si všimnout, že struktura kódu je nyní naprosto odlišná a nebýt řetězců, nebylo by v žádném případě možné poznat, že dekompilovaný kód může být podobný původnímu.

Nedostatky dekompilovaného kódu zde však nejsou způsobeny nekvalitním zpětným překladem, ale optimalizacemi a dalšími fázemi překladače. Kód informace o původní struktuře neobsahoval ani v jazyce symbolických instrukcí (a většinou již ani v optimalizovaném vnitřním kódu překladače) a tudíž nelze lépe rekonstruovat.

Kapitola 5

Původní stav

Obsah této kapitoly je klasifikován jako utajený, viz licenční ujednání.

Kapitola 6

Návrh nového řešení

Obsah této kapitoly je klasifikován jako utajený, viz licenční ujednání.

Kapitola 7

Implementace

Obsah této kapitoly je klasifikován jako utajený, viz licenční ujednání.

Kapitola 8

Testování

Obsah této kapitoly je klasifikován jako utajený, viz licenční ujednání.

Kapitola 9

Vytvořený nástroj

Obsah této kapitoly je klasifikován jako utajený, viz licenční ujednání.

Kapitola 10

Závěr

V úvodu této práce byl čtenář seznámen se základy zpětného inženýrství, zejména problémy, kterým čelí zpětný překlad binárního souboru. Byl také seznámen s projektem *LLVM*, do kterého patří i překladač *Clang*, jehož knihovny jsou využívány v této práci.

Cílem této bakalářské práce bylo navrhnout, vyvinout, otestovat a nasadit do provozu nástroj *csim*, sloužící pro porovnání dvou zdrojových souborů, určený primárně pro testování kvality zpětného překladače *Retargetable Decompiler* společnosti *AVG Technologies s.r.o.* Nástroj tak musel být schopný nahradit stávající – tedy být obsáhlejší, přesnější a rychlejší.

Jak je patrné z předchozí kapitoly, tyto vytyčené cíle se podařilo splnit. Nástroj je až několikanásobně rychlejší než původní, a to i přesto že analyzuje soubory dle více kritérií. Tyto kritéria navíc sdružuje do analýz, u kterých nabízí i celkový výsledek a také výslednou podobnost souborů (což původní nástroj neuměl). Poskytuje také lépe čitelné výstupy ve více formátech (včetně formátu pro strojové zpracování). Jeho součástí jsou i sady automatizovaných testů, testující proces porovnání i jednotlivé poskytované výsledky.

Nástroj již v nočních testech, ve kterých se nachází jeho primární využití, nahradil původní. Provoz byl již zprvu bezproblémový. Bude tak nadále využíván při vývoji zpětného překladače.

Mimo to byl nástroj vyvíjen s ohledem na dobré programátorské praktiky a je tedy snadno rozšiřitelný, což byl také jeden z požadavků zadání. Nástroj tak může reagovat na průběh vývoje zpětného překladače přidáváním dalších testovacích kritérií (a změny lze snadno ověřit díky dostupnosti automatizovaných testů).

I když bylo primárním účelem nástroje testování, algoritmy byly psány obecněji a byly analyzovány také pro možný přínos při odhalování škodlivého softwaru, s čímž souvisí podpora detekce optimalizace zvané *inlining* nebo naopak detekce rozdělení funkcí. I v tomto ohledu nástroj ukázal jistý potenciál. Navíc tyto algoritmy měly také příznivý vliv na primární účel – testování.

Celkově byly splněny všechny požadavky kladené na výsledky této práce, ty byly navíc ještě rozšířeny. I přesto by bylo možné nástroj dále vylepšovat, zejména v oblasti detekce škodlivého softwaru a správnosti promítání podobných funkcí se stejnou polohou v grafu volání. Nástroj by také bylo možno rozšířit o analýzy konstrukcí specifických pro jazyk C++, jehož podpora zpětným překladačem je do budoucna plánována.

Literatura

[1] Choosing the Right Interface for Your Application. [online]. [cit. 2016-01-20]. Dostupné z: http://clang.llvm.org/docs/Tooling.html.

[2] Clang: A C language family frontend for LLVM. [online]. [cit. 2016-01-20]. Dostupné z: http://clang.llvm.org.

[3] Clang vs Other Open Source Compilers. [online]. [cit. 2016-01-20]. Dostupné z: http://clang.llvm.org/comparison.html.

[4] DiffNow - compare files online. [online]. [cit. 2016-04-28]. Dostupné z: https://www.diffnow.com/.

[5] GCC Optimization options. [online]. [cit. 2016-04-10]. Dostupné z: http://www.gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[6] Google Code Archive: py2llvm. [online]. [cit. 2016-05-10]. Dostupné z: https://code.google.com/archive/p/py2llvm.

[7] LLVM Clang 3.7 vs. GCC Compiler Benchmarks on Linux. [online]. [cit. 2016-04-27]. Dostupné z: http://www.phoronix.com/scan.php?page=Misc&item=clang-37-gcc52.

[8] The LLVM Compiler Infrastructure. [online]. [cit. 2016-01-20]. Dostupné z: http://llvm.org.

[9] Retargetable Decompiler. [online]. [cit. 2016-01-20]. Dostupné z: https://retdec.com.

[10] Themida: Advanced Windows Software Protection System. [online]. [cit. 2016-05-10]. Dostupné z: http://www.oreans.com/themida.php.

[11] zynamics.com - BinDiff. [online]. [cit. 2016-04-27]. Dostupné z: https://www.zynamics.com/bindiff.html.

[12] Aho, A. V.; Lam, M. S.; Sethi, R.; aj.: *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, 1986, ISBN 0201101947, 303–310 s.

[13] Chikofsky, E.; Cross, J.: Reverse engineering and design recovery: a taxonomy. *IEEE Software*, ročník 7, č. 1, 1990: s. 13–17, ISSN 0740-7459.

[14] Cifuentes, C.; Gough, J. K.: Decompilation of Binary Programs. *Software: Practice and Experience*, ročník 25, č. 7, 1995: s. 811–829, ISSN 0038-0644.

[15] Eagle, C.: *The IDA Pro Book*. San Francisco: No Starch Press, druhé vydání, 2011, ISBN 1593272898, 127–166 s.

[16] Eilam, E.; Chikofsky, E.: *Reversing: secrets of reverse engineering.* Indianapolis: Wiley Publishing, Inc., 2005, ISBN 9780764574818.

[17] Gosling, J.; McGilton, H.: *The Java language Environment.* Sun Microsystems, 1996.

[18] Hamilton, J.; Danicic, S.: An Evaluation of Current Java Bytecode Decompilers. *University of London, UK*, 2008.

[19] Johnson, M.: Intermediate Representation. Sylabus, 2010.

[20] Křoustek, J.: Retargetable Analysis of Machine Code. 2015: str. 190.

[21] Křoustek, J.; Matula, P.; Zemek, P.; aj.: A Novel Approach to Online Retargetable Machine-Code Decompilation. *Journal of Network and Innovative Computing*, ročník 2, č. 1, 2014: s. 224–232.

[22] Li, X.; Loh, P. K. K.; Tan, F.: Mechanisms of Polymorphic and Metamorphic Viruses. *European Intelligence and Security Informatics Conference*, 2011.

[23] Meduna, A.: *Elements of Compiler Design.* Auerbach Publications, 2007, ISBN 1439815674.

Přílohy

Seznam příloh

Příloha A

Obsah DVD

Přiložené DVD obsahuje ve veřejné verzi bakalářské práce pouze tuto práci ve formátu PDF a její zdrojové kódy v LaTeXu.

- `/thesis.pdf` – tato práce ve formátu PDF,
- `/thesis/` – zdrojové texty práce v LaTeXu.